Accordion Clocks: Logical Clocks for Data Race Detection

Mark Christiaens and Koen De Bosschere

Ghent University, Department ELIS
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{mchristi,kdb}@elis.rug.ac.be

Abstract. Events of a parallel program are no longer strictly ordered as in sequential programs but are partially ordered. Vector clocks can be used to model this partial order but have the major drawback that their size is proportional to the total number of threads running in the program. In this article, we present a new technique called 'accordion clocks' which replaces vector clocks for the specific application of data race detection. Accordion clocks have the ability to reflect only the partial order that is relevant to data race detection. We have implemented accordion clocks in a Java virtual machine and show through a set of benchmarks that their memory requirements are substantially lower than for vector clocks.

1 Introduction

Parallel systems have a notion of time which differs considerably from that of sequential programs. In sequential programs, all operations are ordered one after the other as they are executed. In parallel programs on the other hand, operations can be performed simultaneously. The total order of sequential programs is lost and replaced by a partial order.

To model this new notion of time, Lamport introduced the 'happened before' relation in [6]. Further investigations into this notion resulted in the definition of a data structure called 'vector clocks' [7,4,5]. Vector clocks exactly model the partial order of the events in a parallel system but have the major disadvantage that a vector timestamp (a 'moment' in vector time) consumes memory proportional to the number of concurrent threads in the system [1]. When dealing with systems with unbounded parallelism the modeling of the happened before relation can be a major problem since the vector clocks grow without limits.

Vector clocks are used in a large number of applications: distributed breakpoints, detection of global predicates, distributed shared memory, etc. In this article we will focus on another application: the detection of data races. A data race occurs when two threads modify a common variable without proper synchronisation. This results in non-deterministic behaviour and can be very hard to spot.

One method to detect data races maintains a set of vector clocks for every shared variable. Using these vector clocks, every access to the shared variables

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 494–503, 2001.

is checked. If the vector clocks show that a modification to a shared variable is not properly synchronized with previous or following accesses then a data race is detected. Since vector clocks are maintained for every shared variable, the problem of the size of the vector clocks cannot be ignored.

In this article we propose a refinement of vector clocks called 'accordion clocks'. An accordion clock can be used as a drop-in replacement for vector clocks. They have the additional property that when a new thread is created, they dynamically grow and when a thread 'dies', their size can sometimes be reduced. By taking into account that for data race detection, the full capability of a vector clock is not needed, an accordion clock has less strict requirements to reduce its size than traditional logical clocks. Accordion timestamps have a memory consumption proportional to the number of 'living' threads. Furthermore, accordion clocks have approximately constant execution time overhead.

We will give a short overview of the concept of vector clocks and more generally logical time and establish a formal framework in Section 2. In Section 3 we show how vector clocks are applied to detect data races and illustrate their inherent limitations. In Section 4 we will define accordion clocks. Finally, in Section 5 we present a number of measurements of a modified Java Virtual Machine which illustrate the performance of accordion clocks. In Section 6 we draw some conclusions.

2 Logical Time

Parallel systems consist of a set of threads, T, executing events from the event set, E. Events can be thought of as being processor instructions or could be a coarser type of operation. The subset of all events executed by thread, $T_i : T$, is $E_i \subset E$. Every event is executed on one thread so we can define $\forall e : E_i.T_{id}(e) = i$. It is clear that all events in E_i are sequentially ordered by the fact that operations are executed sequentially on a processor. We can therefore define a relation on the set E_i, $\overset{seq}{\rightarrow}$:

$$\forall(a,b) : {E_i}^2.(a \overset{seq}{\rightarrow} b \equiv \text{a is executed before b}) \tag{1}$$

Things are not so clear when dealing with events executed by different threads. Since there is usually no common clock to keep the two threads exactly synchronized, we cannot use 'real time' to order these events. Among different threads, the only order between events we can be sure of is between events that perform some sort of message exchange. Therefore we define a second relation on E:

$$\forall(a,b) : E^2.(a \overset{msg}{\rightarrow} b \equiv \text{a sends message to b}) \tag{2}$$

Using $\overset{seq}{\rightarrow}$ and $\overset{msg}{\rightarrow}$, we can build the 'happened before' relation (also called the 'causal relation'), $\overset{cau}{\rightarrow}$, as defined by Lamport in [6]:

$$\forall(a,b) : E^2.(a \overset{cau}{\rightarrow} b \equiv (a \overset{seq}{\rightarrow} b) \vee (a \overset{msg}{\rightarrow} b) \vee (\exists c : E.a \overset{cau}{\rightarrow} c \wedge c \overset{cau}{\rightarrow} b)) \tag{3}$$

Intuitively, it orders all events that *could* have influenced each other during the execution of the parallel system. It is also called the 'causal relation'. Two events are said to be parallel when they are not ordered by the causal relationship:

$$\forall(a,b) : E^2.(a \parallel b \equiv \neg(a \overset{cau}{\rightarrow} b \vee b \overset{cau}{\rightarrow} a)) \tag{4}$$

Many data structures have been constructed to detect part or whole of the causal relationship and are called 'logical clocks'. Logical clocks provide a mapping, $L : E \rightarrow D$, from events to 'timestamps', from set D, such that some or all properties of the causal relationship are carried over onto a relation of the timestamps, $<$.

One such mapping, V_c, is a vector clock [7,4]. A vector clock timestamp is an n-tuple of dimension equal to the number of threads in the parallel system, #T. With $\square n \equiv \{i : \mathbb{N} \mid i < n\}$ a subrange of the natural numbers, $I \equiv \square(\#T)$ the set of all thread identifiers, $V \equiv I \rightarrow \mathbb{N}$ the set of all vector clock timestamps and $\delta_{i,j}$ the Kronecker symbol, the mapping, V_c, of events to vector clock timestamps becomes:

$$\begin{gathered} V_c : E \rightarrow V \\ \forall i,j : I^2.\forall a : E_i.V_c(a)[j] = \max(\{V_c(b)[j] \mid b : E.b \overset{cau}{\rightarrow} a\} \cup \{0\}) + \delta_{i,j} \end{gathered} \tag{5}$$

A property of this mapping is that we can define a relation, $<$, on the resulting timestamps that reflects exactly the causal relation:

$$<\equiv (v_1, v_2) : V^2.(\forall i : I.v_1[i] \leq v_2[i]) \wedge v_1 \neq v_2 \tag{6}$$

i.e. one timestamp is 'smaller' than the other if all components are smaller or equal and the two timestamps are not identical.

The causal relationship is carried over exactly to the vector clock timestamps.

$$a \overset{cau}{\rightarrow} b \equiv V_c(a) < V_c(b) \tag{7}$$

If we know the threads, T_i and T_j, which executed the events, a and b, then this comparison can be optimized as follows.

$$\forall(a,b) : (E_i \times E_j).((i \neq j) \Rightarrow (a \overset{cau}{\rightarrow} b \equiv V_c(a)[i] \leq V_c(b)[i])) \tag{8}$$

It has been proven in [1] that the size of a vector clock timestamp for general programs must at least be equal to the number of parallel threads in the system. This poses some serious scalability issues in highly multi-threaded applications.

In what follows, we will describe a new technique, called 'accordion clocks', which alleviates the problem of the growing size of timestamps for the specific application of data race detection.

3 Data Race Detection

Using the happened before relation, we can define a datarace more formally. If we define $R(e)$ as the set all locations read during event e and similarly $W(e)$ as

the set of all locations written to during event e then a data race between two events, e_1 and e_2, can be defined more formally as:

$$Race(e_1, e_2) \equiv \begin{cases} e_1 \parallel e_2 \\ (R(e_1) \cap W(e_2)) \cup (W(e_1) \cap R(e_2)) \cup (W(e_1) \cap W(e_2)) \neq \emptyset \end{cases} \tag{9}$$

Several approaches to detecting data races were described in [9,3,8,10]. The approach we will focus on consists of maintaining 'access histories' as described in [3]. An access history of a memory location consists of a list of previous read and write operations to this location. It contains all the most recent read operations that are not causally ordered with one another and the last write operation that occurred to this location.

Each time a new read or write operation is performed, this new operation is checked with the access history. If Condition 9 holds between the new read/write event and an event in the access history, a data race is reported. Afterwards, the read or write operation is recorded in the access history, possibly removing other events already present.

Since every access history can contain as many as $\#T + 1$ vector clock timestamps, there can be a very large number of vector timestamps present in the system. Each of these grows proportional to the number of threads. We can clearly conclude that this approach to data race detection does not scale well with an increasing number of threads.

4 Accordion Clocks

The basic idea for accordion clocks comes from the observation that usually not all threads are active at the same time. Consider the behaviour of a simple FTP-server as seen in Figure 1. We see that for each file request made to the server, a new slave thread is created. Depending on the size of the file to transfer, the bandwidth of the network connection to the client and many other factors, this slave thread can take a variable amount of time to finish. When the slave thread has finished transferring the file, it is destroyed.

Clearly, over a long period of time, only a small fraction of the slave threads will run concurrently. Still, the size of the vector clock will be equal to the total number of threads executed by the FTP-server. Indeed, if we would try to remove the information about the execution of thread T_2 when the thread has finished, we would run the risk of no longer detecting all data races.

Take for example event e_1 on thread T_2. Even when thread T_2 has finished execution, this event can still cause a race with another event e_2 on thread T_3. Indeed, if T_2 had run a little slower, e_2 would have been executed before e_1, which clearly constitutes a data race. The cause for this data race is that T_2 simply dies when it has finished its task without synchronizing with any other thread. Therefore, the risk for data races with event e_1 will continue to exist indefinitely.

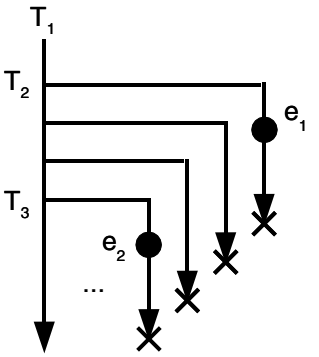

Fig. 1. A simple FTP-server spawning threads for each file request

Using vector clocks, we are stuck here. In general, we cannot reduce the size of a vector clock. The best we can do is increase the size of the vector clocks dynamically as new threads are created.

The Equation 8 shows us how to improve vector clocks. Suppose we have two events, e_i and e_j, from resp. threads, T_i and T_j. If we want to know whether these two events were executed in parallel then, according to Definition 4, we must verify that neither $e_i \overset{cau}{\rightarrow} e_j$ nor $e_j \overset{cau}{\rightarrow} e_i$ holds. To perform this check, Equation 8 tells us we only need the values of the vector clocks at indices i and j. Or, if we invert this reasoning, we can throw away all positions in the vector clocks corresponding to threads from which we have no events left that interest us when doing data race detection!

The events that still interest us are stored in the access histories of the shared variables. These events are gradually being removed by the following three mechanisms:

1. When a read operation occurs on a shared variable, its access history is updated. All the read operations that are causally ordered before the new read operation are removed. Then the new read operation is added. See [3].
2. When a data structure is destroyed and its memory is freed, there can no longer be data races on this data structure. There is therefore no more need for its access history. As a consequence, the access history can be destroyed.
3. Periodically, for example after every garbage collection, we can perform a 'cleanup' as follows. We take a consistent cut, c of the system from the set of consistent cuts CC, i.e. we stop the system in a consistent state. ($\mathcal{P}$ is the 'powerset', the set of all subsets of a set)

$$CC = \{c : \mathcal{P}(E) \mid \forall e_1 : c . \forall e_2 : E . e_2 \overset{cau}{\rightarrow} e_1 \Rightarrow e_2 \in c\} \tag{10}$$

For every thread, T_i, we look up the event it is currently executing, $\mathrm{cur}(i, c)$, at the consistent cut, c. We combine the vector timestamps of each of these events to calculate the 'past' vector timestamp, $\mathrm{past} : CC \rightarrow V$, of all these

threads as follows.

$$\forall i : I.\forall c : CC.\text{past}(c)[i] \equiv \min\{V_c(\text{cur}(j,c))[i] \mid j \in I\} \tag{11}$$

All events present in the access history that have occurred before past(c) can no longer be parallel with any new events that will occur. This means that no race can occur between these old events and any new events. We can therefore safely remove these from the access history.

Due to the three mechanisms described above, the access histories at a consistent cut, $c : CC$, contain only a subset of all the events executed by the threads. If we let $S(c) \subset E$ be this set and if we let the predicate *running* indicate that a thread is currently running in the system then we can construct the set of active threads at a consistent cut, $T_{act}(c) \subset T$ as follows

$$\forall c : CC.T_{act}(c) \equiv \{t_i : T \mid \exists e : S(c).T_{id}(e) = i\} \cup \{t : T \mid running(t)\} \tag{12}$$

The threads that are not member of the active set have either not run yet or have run but all traces of their activities have disappeared from the access histories. If a thread has not run yet, no events performed by this thread can be involved in a data race yet so no room for it must yet be allocated in the timestamps of events. If a thread has run, but none of the events performed by it are stil 'remembered' in the access histories, again no room must remain allocated for it in the timestamps.

Either way, these threads can currently not be involved in a data race. We will not need to compare the vector timestamps of events from threads in this set with each other so there is no need to maintain the indices in the vector timestamps for these threads.

As threads enter and leave the active set, we can adjust the vector clocks accordingly. When a thread has not yet run, and is about to start, we can easily expand the existing vector clock timestamps to include a 0 at the index corresponding to the new thread. If a thread has finished its execution and none of its events are present in the access histories, then we can remove the corresponding index from all timestamps and vector clocks.

If we return to our example from Figure 1 we see that, although with vector clocks we can never remove or reuse the index corresponding to thread T_2 from the vector clocks because T_2 fails to synchronize at the end of its life, removing an index is possible with the above aproach. Suppose that event e_1 accessed a file descriptor that was used to read a file. When T_2 terminates, it frees the file descriptor (by itself or through a garbage collector). At this point, the access history of the file descriptor is also removed and e_1 is no longer of importance. If the other events performed by T_2 are also removed from the access history, then all knowledge of T_2 can be dropped from the timestamps.

Using this insight, we can define a logical clock which grows when threads are started and shrinks when all events of a thread are 'forgotten' by the other threads. We call this logical clock an 'accordion clock'.

An accordion clock, A_c, is a mapping from an event, $e : E$, to an accordion clock timestamp, $a : A$, at a consistent cut, $c : CC$, in the execution of the

system. An accordion clock timestamp is a pair. The first component of the pair is denoted as a_s. It itself is a tuple of scalar values that indicate the local time, just like the values of the vector clock time stamp. The second component of the pair is a thread identifier, $a_{id} : I$, indicating the thread that performed the event for which this time stamp was generated.

$$\begin{gathered} A = \mathcal{T} \times I \\ A_c : (E, CC) \rightarrow A \\ \forall e : E. \forall c : CC. A_c(e, c) = (S_{ac}(e, c), T_{id}(e)) \end{gathered} \tag{13}$$

$$\begin{gathered} S_{ac} : (E, CC \ni c) \rightarrow T_{act}(c) \rightarrow \mathbb{N} \\ \forall c : CC, \forall e : E. \forall i : T_{act}(c). S_{ac}(e, c)[i] = V_c(e)[i] \end{gathered} \tag{14}$$

In essence, an accordion clock timestamp contains the same scalar values as the vector clock timestamp but only those we are still interested in at a certain point in the execution of the system. The timestamp therefor requires only $O(\#T_{act}(c))$ memory.

The comparison of accordion clock timestamp, $<_{ac}$, at a certain consistent cut, $c : CC$, becomes

$$<_{ac} \equiv (a_1, a_2) : A^2. a_{1,s}[a_{1,id}] \leq a_{2,s}[a_{1,id}] \tag{15}$$

5 Implementation and Measurements

We have implemented accordion clocks in a race detection system called 'TRaDe' [2]. TRaDe is built on a Java virtual machine from Sun that is adapted to on-the-fly detect data races in multi-threaded Java programs that are being executed on the virtual machine. The definition of data races given in Formula 9 is used. The accordion clocks are used to avoid the size problem of vector clocks.

We have used Condition 14 to check whether an accordion clock's size could be reduced. We performed this check after every garbage collection of the JVM. Since the garbage collector requires that the JVM be halted temporarily, this seems an appropriate time to perform our analysis.

To test the performance of the accordion clocks we used the following set of benchmarks.

In Figure 5 we see the results of using vector clocks (on the left) vs. accordion clocks (on the right). In each of the graphs, we show the amount of memory consumed by the data structures needed for performing data race detection. We split the memory consumption into two parts. At the top of each graph we see the memory used to create resp. the vector clocks and accordion clocks in the system. At the bottom, we see the remaining memory used for storing, among others, the access histories. The memory used by the virtual machine itself is not shown. All other optimisations which reduce the size and number of the access histories are enabled during both executions. Only the improvement in memory consumption due to the use of accordion clocks is measured.

On the left of each of the figures, we see that in general the vector clocks continuously grow and tend to consume a large amount of memory after only a

Table 1. Description of the Java benchmarks used

Name	Description
SwingSet Demo	A highly multi-threaded demo, included with the JDK 1.2, of the Swing widget set. It demonstrates buttons, sliders, text areas, ... Every tab in the demo was clicked twice during our tests. Immediately thereafter it was shut down.
Java2D Demo	A highly multi-threaded demo, included with the JDK 1.2. It demonstrates the features of the Java 2D drawing libraries and is highly multithreaded. Every tab in the demo was clicked 5 times. Immediately thereafter it was shut down.
Forte for Java	A full-blown Java IDE with object browsers, visual construction of GUI code, ... To exercise the benchmark, we created a small dialog while doing race detection.

few minutes of performing data race detection. Clearly, this behaviour prohibits the use of vector clocks for doing data race detection for a long period of time.

On the right, we show the use of accordion clocks. For the Java2D and Forte benchmarks we see an almost ideal behaviour. Originally the vector clocks grew proportional with time. This memory consumption is reduced to a constant cost by using accordion clocks. What is more, we have succeeded in leveling off the memory consumption for data race detection since the other data structures apparently, after a startup period, also consume a constant amount of memory.

We have also added a figure of the memory consumption of the Swing benchmark. Again, we can see that the memory needed to maintaining accordion clocks is substantially smaller than for vector clocks. Here we do not yet succeed in reducing memory consumption for data race detection to a constant cost. Both the vector clocks and the access histories grow proportionally with time. Nevertheless, using accordion clocks, we at least succeed in reducing considerably the memory consumption for the clocks. The cause of the memory increase in this benchmark is not yet clear to us. Further research will be needed.

The time overhead of using accordion clocks is negligable. On the one hand, some overhead is incurred by reducing and increasing the size of the accordion clocks. On the other hand, since accordion clocks are usually substantially smaller than vector clocks, the cost of comparing, copying, etc. of their data structures is reduced. Time overhead of the accordion clocks is currently not a real concern since the time overhead incurred by performing the checks of read and write operations for data race detection is by far the predominant factor in the overhead.

6 Conclusions

In general, the maintenance of vector timestamps consumes memory proportional to the number of threads in a program. In this article, we have shown

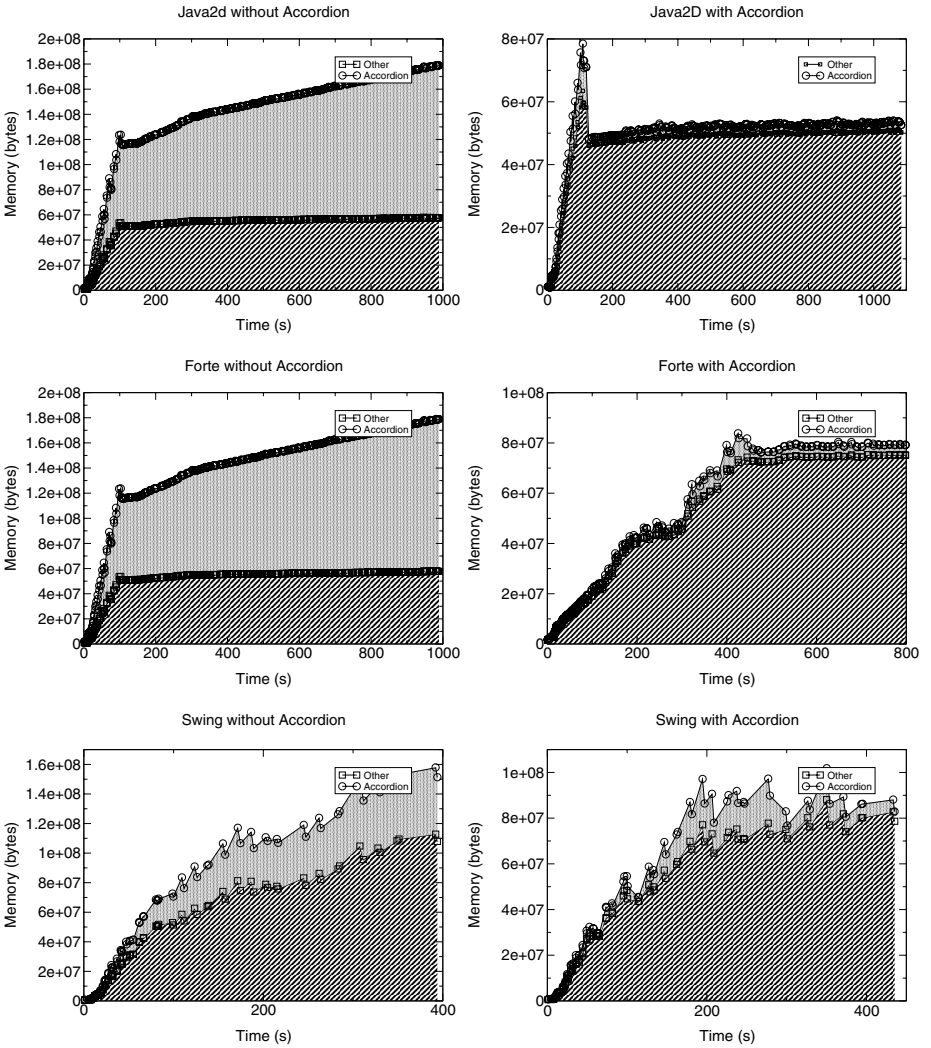

Fig. 2. Memory consumption of data structures for data race detection for the three benchmark programs used

that by taking into account what the vector timestamps will be used for, the size of timestamps can be reduced beyond the limit of the total number of threads running in the system. We have defined a new type of logical clock that has the ability to grow and shrink, called an accordion clock. An accordion clock has the same order of magnitude of execution time consumption as a vector clock. It has the potential to reduce the memory consumption considerably by adapting dynamically the size of the timestamps when it can be verified that no comparison between events of certain threads will be necessary in the future.

Acknowledgements

Mark Christiaens is supported by the IWT SEESCOA project (IWT/ADV/ 980.374). We are grateful to Michiel Ronsse for proofreading this article and his many stimulating remarks.

References

1. Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 1(39):11–16, July 1991. 494, 496
2. Mark Christiaens and Koen De Bosschere. Trade, a topological approach to on-the-fly race detection in java programs. In *Proceedings of the Java Virtual Machine Research and Technology Symposium 2001*, pages 105–116, Monetery, California, USA, April 2001. USENIX. 500
3. A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10, March 1990. 497, 498
4. C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and distributed debugging*, pages 183–194, May 1988. 494, 496
5. Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *21st Annual Hawaii International Conference on System Sciences*, volume II, pages 166–175. IEEE Computer Society, January 1988. 494
6. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. 494, 495
7. Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Intl. Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., North-Holland, 1989. 494, 496
8. Michiel Ronsse. *Racedetectie in Parallelle Programma's door Gecontroleerde Heruitvoering*. PhD thesis, Universiteit Gent, May 1999. 497
9. Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999. 497
10. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Operating Systems Review*, volume 31, pages 27–37. ACM, October 1997. 497